REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

REPORT SECURITY CLASSIFICATION classified	1b. RESTRICTIVE MARKINGS
SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
DECLASSIFICATION/DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
NAME OF PERFORMING ORGANIZATION val Postgraduate School	6b. OFFICE SYMBOL (*if applicable*) 32
7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
ADDRESS (City, *State, and ZIP Code*) nterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
NAME OF FUNDING / SPONSORING RGANIZATION	8b. OFFICE SYMBOL (*if applicable*)
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
ADDRESS (City, *State, and ZIP Code*)	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO	PROJECT NO	TASK NO	WORK UNIT ACCESSION NO

TITLE (*Include security Classification*)
ltiple-Valued Programmable Logic Array Minimization By Concurrent Multiple and Mixed Simulated Annealing

PERSONAL AUTHOR(S) Cem Yildirim

a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15. PAGE COUNT
ster's Thesis	FROM _____ TO _____	1992 December 10	60

SUPPLEMENTARY NOTATION
e views expressed in this thesis are those of the author and do not reflect official policy or position of the Department of Defense or the U.S. Government

COSATI CODES			18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
ELD	GROUP	SUB-GROUP	Multi-Valued Logic; Minimization Heuristics; PLA Design; Simulated Annealing

ABSTRACT (*Continue on reverse if necessary and identify by block number*)

The process of finding a guaranteed minimal solution for a multiple-valued programmable logic expression requires an exhaustive search. Exhaustive arch is not very realistic because of enormous computation time required to reach a solution. One of the heuristics to reduce this computation time and ovide a near-minimal solution is simulated annealing.

This thesis analyzes the use of loosely-coupled, course-grained parallel systems for simulated annealing. This approach involves the use of multiple pro-ssors where interprocess communication occurs only at the beginning and end of the process. In this study, the relationship between the quality of solu-n, measured by the number of products and computation time, and simulated annealing parameters are investigated. A simulated annealing experiment also investigated where two types of moves are mixed. These approaches provide improvement in both the number of product terms and computation me.

DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☐ UNCLASSIFIED/UNLIMITED ☐ SAME AS REPORT ☐ DTIC USERS	Unclassified

NAME OF RESPONSIBILE INDIVIDUAL	22b. TELEPHONE (*Include Area Code*)	22c. OFFICE SYMBOL
ller, Jon T.	(408) 646-3299	EC/Bu

T259324

Approved for public release; distribution is unlimited.

Multiple-Valued Programmable Logic Array Minimization By Concurrent Multiple and Mixed Simulated Annealing

by

Cem Yıldırım
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1985

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 1992

ABSTRACT

The process of finding a guaranteed minimal solution for a multiple-valued programmable logic expression requires an exhaustive search. Exhaustive search is not very realistic because of enormous computation time required to reach a solution. One of the heuristics to reduce this computation time and provide a near-minimal solution is simulated annealing.

This thesis analyzes the use of loosely-coupled, course-grained parallel systems for simulated annealing. This approach involves the use of multiple processors where interprocess communication occurs only at the beginning and end of the process. In this study, the relationship between the quality of solution, measured by the number of products and computation time, and simulated annealing parameters are investigated. A simulated annealing experiment is also investigated where two types of moves are mixed. These approaches provide improvement in both the number of product terms and computation time.

c.1

TABLE OF CONTENTS

I. INTRODUCTION 1

A. MOTIVATION 1

B. BACKGROUND 4

II. NEW APPROACHES FOR SIMULATED ANNEALING 7

A. DIFFERENT PATHS 7

B. ANNEALING WITH MIXED MOVES 10

C. CONCURRENCY IN SIMULATED ANNEALING WITH MULTIPLE AND MIXED MOVES 12

III. OPTIMUM PARAMETERS FOR CONCURRENT AND MIXED SIMULATED ANNEALING 16

IV. EXPERIMENTAL RESULTS 19

V. CONCLUSIONS 25

VI. APPENDIX A: FLOW DIAGRAM OF THE ALGORITHM 27

V. APPENDIX B: C CODE UTILIZED 29

LIST OF REFERENCES 50

INITIAL DISTRIBUTION LIST 52

ACKNOWLEDGEMENT

I would like to express my sincere appreciation to the Turkish Navy and United States Navy for providing this exceptional educational opportunity. In appreciation for their time and effort, many thanks go to the staff and faculty of the Electrical and Computer Engineering Department, NPS. A special note of thanks goes to Dr. Yang for his support and guidance. I would like to offer special thanks to my advisor Dr. Butler for his guidance and encouragement.

I. INTRODUCTION

A. MOTIVATION

In the last ten years, significant progress has been made in realizing large logic circuits in silicon using very-large-scale-integration (VLSI) technology. With this progress there have been two major problems, interconnection and pin limitation. Indeed, they have become bottlenecks to further integration. Multiple-Valued-Logic (MVL) offers a solution. In MVL, there are more than two levels of logic. MVL has found application in programmable logic arrays (PLA) implemented in charge-coupled devices (CCD) [Ref. 1,2,3,4] and current mode CMOS [Ref. 16,17].

Several heuristic algorithms have been developed related to computer-aided design and logic synthesis tools for multiple-valued PLA's [Ref. 5,6,7,8,9,14,15,16,17]. None of these heuristic algorithms is consistently better than the others [Ref. 9], each having an advantage in specific examples [Ref. 5,6,7,8,9]. Heuristic algorithms are important because algorithms that guarantee a minimal solution require exhaustive search. Exhaustive search is not very realistic due to the immense amount of computation time required to reach a solution.

A new heuristic, simulated annealing, offers a solution for obtaining near-minimal solutions to combinatorial optimization problems with reasonable computation times. Advantages of this technique include its potential to find true minimal solutions, general applicability, and ease of implementation [Ref. 12].

Because of the similarity to the statistical mechanical model of annealing in solids, this technique is called "simulated" annealing. That is, the slow cooling of certain solids results in a state of low energy, a crystalline state, rather than an amorphous state that results from fast cooling.

Simulated annealing is a search of the solution space in a combinatorial optimization problem. Its goal is to find a solution of minimum cost with the repeated application of the following three steps [Ref. 12].

1) **Create a new solution from the current solution:** In this step, the algorithm chooses a pair of product terms, and tries to make a move to reach to a new solution. If the two product terms can be combined into one, they are combined. This is called a cost decreasing move. If they can not be combined, the two are replaced by two or more equivalent product terms. If indeed, there are two new product terms a zero-cost move is proposed. If there are three or more product terms, a cost increasing move is proposed.

2) **Calculate the cost of the new solution:** This step calculates the cost of the move by comparing the number of product terms before and after the move.

3) **If the increase in cost of the new solution is below some specified threshold, it becomes the current solution:** The algorithm decides if the move is accepted or rejected, depending on the threshold which is a function of the cost and the current temperature.

Unlike other minimization techniques (which are classified as direct-cover methods), simulated annealing manipulates product terms directly, breaking them

up and joining them in different ways to reduce the total number of product terms. Manipulation of product terms is done nondeterministically. That is, randomly chosen product terms are randomly combined (cost decreasing move), reshaped or divided (cost increasing moves). As with the mechanical applications of annealing, the solution set is heated first and than allowed to cool slowly in order to reach a crystalline state or optimum solution. Given an expression in the form of a set of product terms, the algorithm [Ref. 11] divides and recombines the product terms, gradually progressing toward a solution with fewer product terms. Although cost increasing moves represent movement away from the optimal solution, they allow escape from local minima. The repeated application of the three steps above usually requires a large number of moves before a minimal or near-minimal solution is achieved. Therefore, a bias is applied which determines the probability of acceptance of cost increasing moves. Initially, the probability of accepting such a cost increasing move is high, between 0.5 and 1.0. However, as the cooling starts, this probability decreases. As a result, at the beginning, wide excursions are made in the solution space, while near the end, only global or local minima explored. When the probability of accepting a cost increasing move is very low, simulated annealing is usually in a global or local minima, which indicates that the probability of escape is also low. Thus, if the probability is decreased quickly (a process called quenching), simulated annealing converges on a local minima with little chance of escape. A preferred approach is to decrease the probability of accepting cost increasing moves

slowly, which means a slow cooling. This allows transitions among more states and improves the chance of finding a global minimum.

A brief review of the minimization problem of an MVL PLA is given below. Then, a new implementation of simulated annealing is presented.

B. BACKGROUND

A product term is expressed as

$$c\ {}^{a_1}x_1^{b_1}\ {}^{a_2}x_2^{b_2} \ldots .\ {}^{a_n}x_n^{b_n} \tag{1.1}$$

, where $c \in \{1,2, \ldots,r-1\}$, is a nonzero constant, where the literal function is given as

$${}^{a_i}x_i^{b_i} = \begin{cases} r-1 & a_i \le x_i \le b_i \\ 0 & otherwise \end{cases} \tag{1.2}$$

and where concatenation is the min function; i.e. $xy=\min(x,y)$. Since the literal function takes on only values *0* or *r-1*, the product of literals is either *0* or *r-1*, while the complete term takes on values *0* or *c*. An *r*-valued function $f(x_1,x_2,\ldots,x_n)$ takes on values from $\{1,2,..,r-1\}$ for each assignment of values to the variables, which are also *r*-valued; i.e. $x_i \in \{0,1,2,\ldots,r-1\}$. A function can be represented by sum of these products as follows

$$\begin{aligned} f(x_1,x_2,\ldots,x_n) = {} & c_1\ {}^{a_{1,1}}x_1^{b_{1,1}}\ {}^{a_{1,2}}x_2^{b_{1,2}} \ldots\ {}^{a_{1,n}}x_n^{b_{1,n}} \\ & +c_2\ {}^{a_{2,1}}x_1^{b_{2,1}}\ {}^{a_{2,2}}x_2^{b_{2,2}} \ldots\ {}^{a_{2,n}}x_n^{b_{2,n}} \\ & +c_3\ {}^{a_{3,1}}x_1^{b_{3,1}}\ {}^{a_{3,2}}x_2^{b_{3,2}} \ldots\ {}^{a_{3,n}}x_n^{b_{3,n}} \\ & + \ldots \quad . \end{aligned} \tag{1.3}$$

,

where + is truncated sum, i.e. $a+b = \max\{a+b , r-1\}$, where the right sum is ordinary addition with a and b viewed as integers.

A minimum sum-of-products expression for $f(x_1,x_2,...,x_n)$ is the one with the fewest product terms. Finding such a solution is based on that shown in Dueck *et al* [Ref. 11]. Given a set of product terms that sum to a given function, the algorithm derives another set by making a move. Similarly, a move is made from the next set, etc. until finally a minimal or near-minimal set is formed. As in [Ref. 11], this thesis investigates two kinds of moves.

1) **Cut-or-Combine:** The two randomly chosen product terms are combined into one, if possible. If not, one is chosen randomly, with probability 0.5. If the chosen product term is a 1 minterm (i.e. a product term of the form $1\ {}^{a_1}x_1^{b_1}\ {}^{a_2}x_2^{b_2}....\ {}^{a_n}x_n^{b_n}$, where $a_i=b_i$ for all i), the current move is abandoned and another pair of product terms is chosen. Otherwise, the chosen product term is divided into two. The process of division occurs either along the logic values or geometrically. If the division is along the logic value, the resulting product terms are the same except for their constant c values which sum to the logic value of product term. If the division is done geometrically, the two product terms have the same constant c value and are adjacent, covering all minterms covered by the original product term.

2) **Reshape:** This move, like Cut-or-Combine, operates on two product terms, and combines the pair if a combine is possible. If not, a consensus term is formed.

If the two product terms overlap, the consensus is the intersection of the two terms with a coefficient that is the truncated sum of the two coefficients. If the two product terms are disjoint (but adjacent), then the consensus is a term taking on two parts of two product terms, with a coefficient that is the minimum of the two product terms. The remaining product terms are chosen so that the fewest number cover all minterms covered by the original product term.

In this thesis, instead of committing to either Cut-or-Combine or Reshape, a mixture of both with different paths is used, and this is performed concurrently in a distributed system. The results of the following three different approaches are investigated:

1) Different paths in the solution space.

2) Mixing types of moves.

3) Concurrency.

These methods are explained in Chapter II. Experimental results of concurrent multiple and mixed simulated annealing are summarized in Chapter III. Concluding remarks are given in Chapter IV.

II. NEW APPROACHES FOR SIMULATED ANNEALING

In the early stages of this thesis, two techniques for the parallelization of simulated annealing for MVL PLA minimization were considered:

1) **Speculative simulated annealing** proposed by Witte *et al* [Ref. 10] : This technique provides speedup only when the cost evaluation time is large compared to the move generation time. This is not the case of our problem, and this approach was abandoned.

2) **The division algorithm** : This algorithm which is an approach to fine-grained parallelism was also abandoned because the original algorithm was efficiently written, and fine-grained parallelism provided only marginal improvement.

As a result of this experience, we concentrated on course-grained parallelism, in which there is minimal communication among processes.

A. DIFFERENT PATHS

During the simulated annealing process, the algorithm randomly walks through the solution space. A random walk through solution space will eventually result in a minimal solution, if there is a nonzero probability of reaching the minimal solution from any other solution. This is one way to achieve a minimal solution with probability approaching 1.0 as time increases. But it is impractical, because of excessive computation time. Simulated annealing uses a different technique for

random walk in which the probability of transition to a higher cost solutions continually decreases toward 0. In this technique, like a random walk, the transition from one solution to another occurs randomly. But unlike random walk, the transition probability is not uniformly distributed among all possible next solutions. It is biased toward lower cost solutions, with the bias increasing as time passes. Fig. 2.1 shows this. All squares in a column represents solutions with the same number of product terms. Columns on the right represent solutions with large

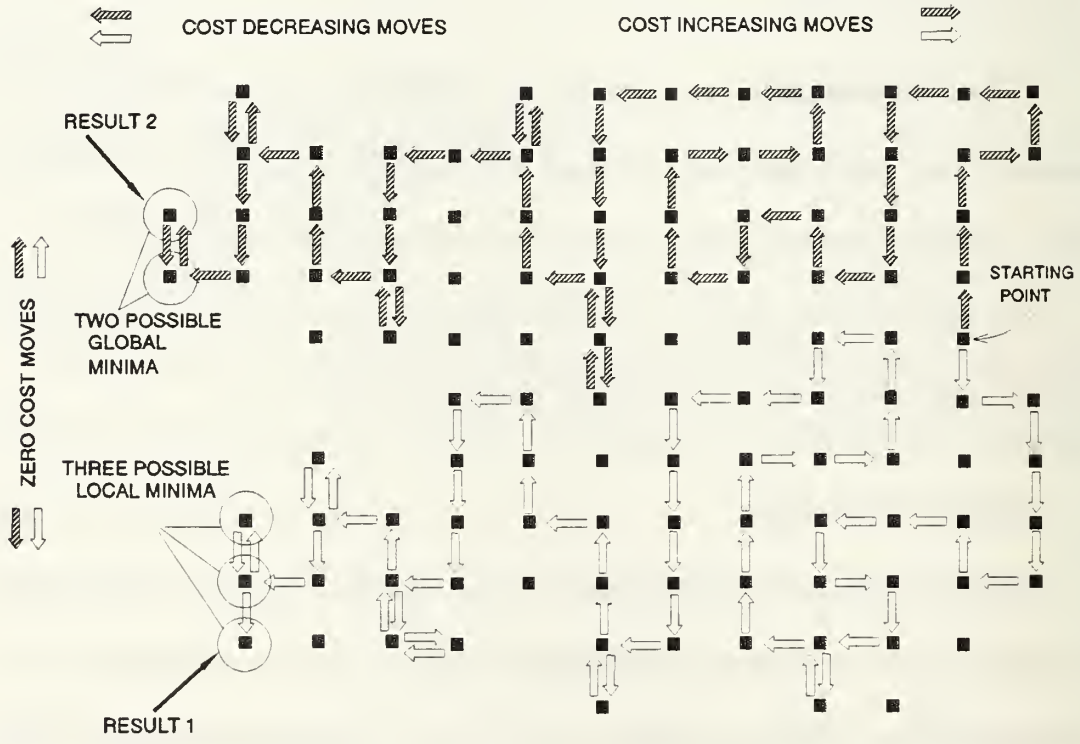


Figure 2.1: Examples of simulated annealing using different paths.

number of product terms. Squares on the extreme left represent local minima. The algorithm can make three types of moves, cost decreasing, cost increasing and zero-

cost moves, represented by right going, left going and vertical arrows, respectively. As the time passes, with the decreasing probability of cost increasing moves, it becomes less likely that a move is made away from the optimal solution. The two paths in Fig. 2.1 shows two examples of simulated annealing experiment. One results in a nonoptimal solution, while the other results in an optimal solution.

Fig. 2.1 also illustrates one approach we considered, different paths. This approach was tested using the Reshape algorithm on ten different functions (C1,C2,..,C10) which were used in Dueck *et al* [Ref. 11]. Each function has 200 minterms. For every function, eight different paths were chosen. Table 2.1 shows the results. The column "FUNCT" shows the different test functions. The columns under the terms "PATH" show the results of the different paths. Each entry shows two results, the number of product terms achieved and the computation time (in parentheses) in seconds. The computation time is CPU seconds on Sun Workstation running the SunOS Release 4.1.1-GFX-Rev2 operating system. The column "OUT" shows the lowest number of product terms. This is considered to be the output of the algorithm. In this column, parentheses enclose the total computation time over the eight paths. Since this was performed on one processor, this total is the computation time for this (sequential) version of the algorithm. The average values of product terms and computation times for each path are given in the second to the last row. As we see from Table 2.1, there is a clear dependence on the path. For example, the eight paths on function C4 yield five different values for a near-minimal solution, 80, 81, 82, 83 and 86 product terms.

TABLE 2.1: SEQUENTIAL MULTIPLE PATHS WITH REGULAR RESHAPE.

Initial Temp : 0.7 Max Valid Factor : 4
Max Frozen : 5 Cooling Rate : 0.93
Max Try Factor : 25

FUNCT	PATH 0	PATH 1	PATH 2	PATH 3	PATH 4	PATH 5	PATH 6	PATH 7	RSAV	OUT
C1	88(59)	86(53)	87(61)	87(56)	87(57)	86(60)	88(56)	90(57)	87.3(57.4)	86(459)
C2	85(63)	86(66)	85(58)	86(57)	86(56)	84(58)	85(63)	86(62)	85.3(60.4)	84(483)
C3	88(59)	89(67)	91(79)	90(58)	89(56)	90(55)	88(65)	89(55)	89.2(61.7)	88(494)
C3	82(60)	83(52)	83(51)	86(52)	80(73)	82(48)	81(54)	82(53)	82.8(55.4)	80(443)
C5	81(51)	82(62)	81(57)	80(62)	79(58)	82(55)	80(52)	80(65)	80.6(57.7)	79(462)
C3	83(54)	81(56)	86(52)	83(55)	82(57)	83(50)	86(61)	82(54)	83.2(54.9)	81(439)
C7	88(57)	88(68)	87(61)	89(57)	87(68)	90(54)	89(58)	87(57)	84.8(60.0)	87(548)
C8	79(54)	79(54)	78(51)	77(48)	81(60)	81(46)	79(49)	79(49)	79.1(51.4)	77(411)
C9	81(55)	83(57)	82(66)	83(56)	83(63)	82(65)	82(56)	83(73)	82.4(61.4)	81(491)
C10	84(64)	84(55)	84(56)	83(60)	81(63)	87(60)	84(63)	83(54)	83.8(59.4)	81(475)
PHAV	83.9(57.6)	84.1(59.0)	84.4(59.2)	84.4(56.1)	83.5(61.1)	84.7(55.1)	84.2(57.7)	84.1(57.9)	84.2(58.0)	**82.4(470.5)**

This experiment shows that there is a slight improvement in the average when we perform eight rather than one simulated annealing experiment. Taking the best from eight yields an average of 82.4 product terms over the ten functions. We can represent the performance of one simulated annealing experiment by averaging the averages for each path in Table 2.1, yielding 84.2. This improvement requires a large computation time, a total of 470.5 seconds vs. 58.0 seconds for one path (calculated by averaging the times for each path). Such a large computation time can be reduced significantly by using multiple processors. Since there is no need to exchange information, the various paths can be executed concurrently. The concurrent execution of this algorithm is discussed in Section C.

B. ANNEALING WITH MIXED MOVES

Results in Dueck *et al* [Ref. 11] show that Reshape produced overall better results than Cut-or-Combine in terms of number of products and computation time.

However, this was not true for every function, as far as number of products is concerned. In a few cases, Cut-or-Combine did better than Reshape.

Starting from this experience, we investigated the use of simulated annealing in which Cut-or-Combine moves were mixed with Reshape moves (Fig. 2.2). For these experiments, different mixture ratios were tried, all of which used Reshape more than Cut-or-Combine. The best results were achieved when Cut-or-Combine moves occurred 4% of the time and Reshape moves occurred 96% of the time. Results are shown in Table 2.2 (with 4% / 96% mixture ratio). This experiment yields an average number of product terms of 81.7 compared with 84.2 for a single experiment using Reshape without Cut-or-Combine.

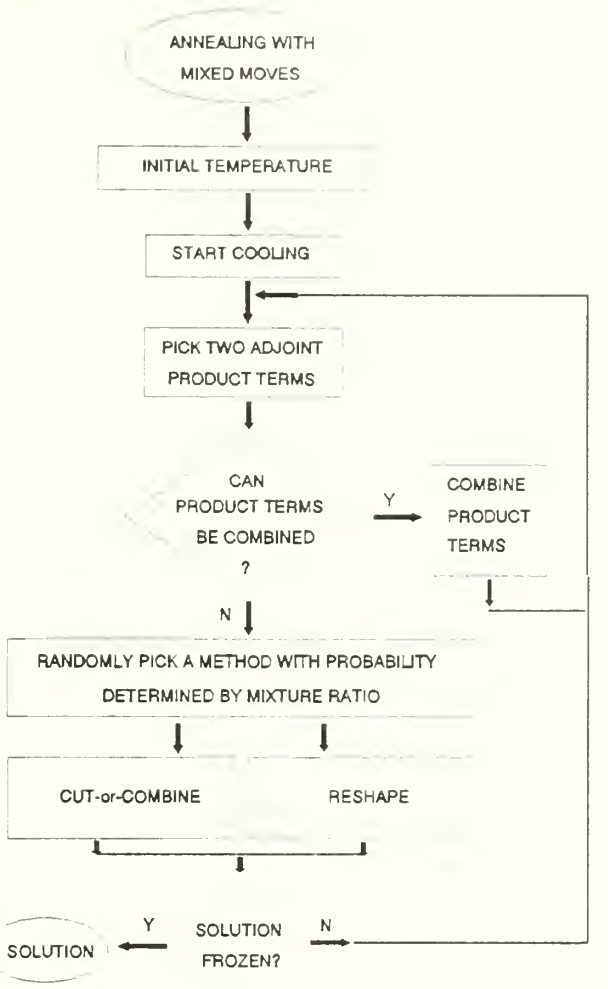


Figure 2.2: Simulated annealing with mixed moves.

TABLE 2.2: SEQUENTIAL MULTIPLE PATHS WITH MIXED ANNEALING.

Initial Temp : 0.6 Max Valid Factor : 4
Max Frozen : 5 Cooling Rate : 0.94
Max Try Factor : 25 Cut-or-Combine Reshape Ratio : 4% 96%

FUNCT	RAVG	PATH 0	PATH 1	PATH 2	PATH 3	PATH 4	PATH 5	PATH 6	PATH 7	OUT
C1	87.3(57.4)	88(57)	87(70)	84(89)	86(70)	85(78)	87(63)	89(93)	90(66)	84(562)
C2	85.3(60.4)	83(66)	85(88)	83(82)	86(91)	84(78)	86(73)	85(96)	85(76)	83(651)
C3	89.2(61.7)	88(97)	89(96)	89(94)	89(77)	89(67)	87(79)	89(79)	89(83)	87(672)
C4	82.8(55.4)	82(65)	82(63)	81(49)	78(53)	83(53)	80(58)	85(62)	81(61)	78(491)
C5	80.6(57.7)	80(91)	79(77)	82(77)	81(67)	82(77)	82(92)	87(79)	80(61)	79(577)
C6	83.2(54.9)	83(70)	82(63)	83(61)	82(96)	86(67)	84(58)	83(61)	84(59)	82(562)
C7	84.8(60.0)	87(61)	86(84)	87(67)	88(99)	86(67)	91(80)	88(89)	88(108)	86(671)
C8	79.1(51.4)	78(67)	79(76)	79(64)	78(61)	78(55)	77(69)	76(61)	79(70)	76(602)
C9	82.4(61.4)	80(87)	84(88)	83(90)	80(67)	83(63)	84(67)	83(65)	83(78)	80(601)
C10	83.8(59.4)	82(73)	86(83)	85(63)	84(75)	83(63)	85(71)	82(73)	84(86)	82(596)
AVRG	84.2(58.0)	83.1(73.4)	83.9(78.8)	83.6(73.6)	83.2(75.6)	83.9(66.8)	84.3(71.0)	84.3(75.2)	84.3(74.8)	**81.7(598.5)**

C. CONCURRENCY IN SIMULATED ANNEALING WITH MULTIPLE AND MIXED MOVES

It has been shown that the use of more than one path with and without mixed moves provides a reduction in the number of product terms over the use of one path. However, substantial computation time is required when one processor is used. But using more than one processor in a distributed system, we can achieve a speedup. To investigate this, eight Sun Workstations were used. Fig. 2.3 shows the program for this distributed system. In the beginning, the host sends the name of the file (in which input function is placed) to the other processors and later assigns itself as processor 0. Since multiple path and mixed simulated annealing is suitable for execution on asynchronous multiple instruction multiple data (MIMD) machines, each workstation can proceed independently of the others. In this algorithm, each processor chooses the paths randomly. Assigning different seeds for the random

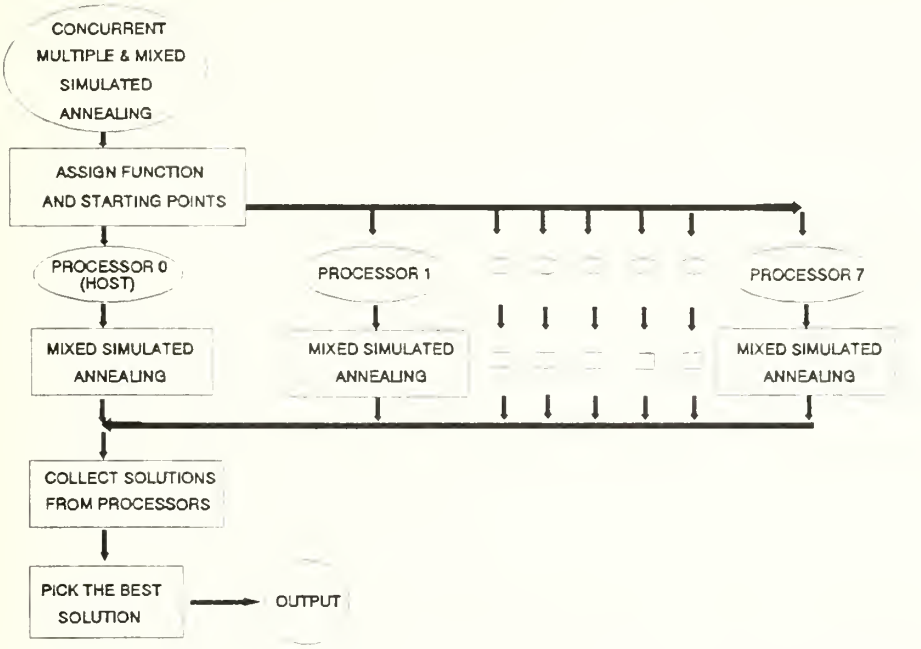


Figure 2.3: Concurrent implementation of multiple path with mixed moves.

number generators of the processors reduces the probability of two processors picking identical next solution states. The probability that two or more processors choose the same pair of product terms can be calculated as

$$1 - \frac{M!}{(M-N)!\ M^N} \quad , \tag{2.1}$$

where

$$M = \binom{P}{2} \quad , \tag{2.2}$$

N is the number of processors used, and P is the number of product terms in the function to be minimized. For instance, with 8 processors and a function with 200 product terms, this probability is 0.0014 . The probability of two processors going to the same next state is even less, since even if they choose the same pair of product

terms, there is a choice of how to divide one of the two. Because of such a small probability, the program does not check if two processors have started with the same next solution state nor if at any point in the computation of the solution is the same. This approach requires the communication only at the beginning and at the end of the process. As soon as the host (processor 0) completes its own assignment, it starts to receive the other processors's results when they are ready to be sent. As the final result of the algorithm, the best output among all processes is chosen.

To compare the results of concurrency, the same tests given in Tables 2.1 and 2.2 were repeated with concurrent and mixed simulated annealing (Test-A and Test-B). In Table 2.3, we see that, the computation time is reduced by factors of 7.1 and 6.6 respectively, which are reasonably close to the theoretical maximum of 8 (there are 8 processors).

TABLE 2.3: TEST COMPARISON

	TEST A	TEST B	TEST C	TEST D	TEST E	TEST F	TEST G	TEST H	TEST I	TEST J
Initial Temp	0.7	0.6	0.6	0.7	0.65	0.62	0.6	0.65	0.7	0.62
Cooling Rate	0.98	0.94	0.94	0.98	0.97	0.91	0.92	0.92	0.91	0.92
Max valid factor	4	4	4	4	5	4	4	4	4	4
Max try factor	25	25	25	25	25	25	25	25	25	20
Max frozen	5	5	5	5	5	5	5	5	5	4
Cut-or-Combine / Resh ratio	no mix	4% 96%	no mix	4% 96%	4% 96%	4% 96%	4% 96%	4% 96%	4% 96%	no mix
Function C1	86(62)	84(93)	85(74)	84(87)	85(186)	84(61)	86(69)	87(75)	85(71)	85(47)
Function C2	84(67)	83(96)	83(70)	83(87)	82(181)	84(59)	83(75)	85(85)	83(74)	84(44)
Function C3	88(80)	87(97)	88(75)	88(89)	86(185)	87(70)	88(74)	89(78)	88(71)	89(52)
Function C4	80(74)	78(65)	80(64)	80(64)	80(175)	80(59)	80(63)	78(65)	81(54)	82(40)
Function C5	79(66)	79(92)	81(73)	79(88)	80(168)	79(59)	80(68)	79(76)	80(60)	81(39)
Function C6	81(62)	82(96)	82(69)	83(69)	80(189)	82(68)	82(74)	83(81)	81(62)	83(43)
Function C7	87(69)	86(108)	87(75)	68(78)	86(162)	86(74)	86(75)	87(79)	86(71)	87(42)
Function C8	77(55)	76(76)	78(69)	77(67)	76(179)	77(58)	77(70)	78(63)	78(53)	77(42)
Function C9	81(75)	80(90)	80(91)	81(87)	79(164)	81(67)	82(78)	81(89)	81(68)	83(43)
Function C10	81(65)	82(86)	82(82)	83(94)	82(210)	83(65)	83(68)	84(74)	83(82)	84(48)
AVERAGE	82.4(67.5	81.7(89.9)	82.6(74.2)	82.4(81.0)	81.6(179.9)	82.3(64.0)	82.7(71.4)	83.1(76.5)	82.6(66.6)	83.5(44.0)

In addition to these experiments, two individual functions, FUNCTION1 and FUNCTION2, were used to compare Cut-or-Combine with Reshape (these were also used in [Ref. 13]). FUNCTION1 is a 4-valued 4-variable symmetric function of 176 minterms with a known minimal solution of 6 product terms. For Cut-or-Combine method, this function is difficult to minimize. FUNCTION2 is a 4-valued 2 variable function for which Reshape can never find a minimal solution.

As we can see from Table 2.4, concurrent Reshape produces the same number of product terms as does Reshape, 7 and 5 for FUNCTION1 and FUNCTION2, respectively. Concurrent Multiple and Mixed produces the best results in both cases, 6 and 4 product terms for FUNCTION1 and FUNCTION2, respectively. This experiment shows that, the occasional application of Cut-or-Combine among many applications of Reshape can produce better results than Reshape alone.

TABLE 2.4: NUMBER OF PRODUCT TERMS AND COMPUTATION TIMES FOR TWO TEST FUNCTION.

	FUNCTION 1			FUNCTION 2		
HEURISTIC	In	Out	Time(sec)	In	Out	Time(sec)
CUT-or-COMBINE	14	19	673	5	4	17.6
RESHAPE	14	7	24.5	5	5	0.25
CONCURRENT RESHAPE	14	7	27.9	5	5	0.43
CONCURRENT MULTIPLE & MIXED	14	6	54.8	5	4*	1.38

*With parameters used in Test-F (Table 2.3).

III. OPTIMUM PARAMETERS FOR CONCURRENT AND MIXED SIMULATED ANNEALING

It is important to consider the effects of various parameters on the performance of the methods discussed above. There are six important parameters in the Concurrent Multiple and Mixed Simulated Annealing algorithm: Mixture rate, cooling rate, initial temperature, maximum valid factor, maximum try factor and maximum frozen factor.

Mixture rate determines the rate how two types of moves, Reshape and Cut-or-Combine are mixed.

Cooling rate controls the number of temperatures that the process sequences through during the transition from the melted to the frozen state.

Initial temperature controls the extent of melting. Since the temperature directly controls the probability of accepting cost increasing moves, a higher initial temperature means a more melted solution.

Maximum valid factor determines the number of moves occurring at each temperature.

Maximum try factor regulates how long the program examines moves prior to continuing on the next temperature. The number of attempted moves for each temperature is calculated by multiplying the maximum try factor by the number of moves.

Maximum frozen factor is used to determine if the process is really in frozen state. This indicates that no more effort should be expended on the expression.

These parameters are tested with the following values.

a. Maximum frozen factor (3 : 4 : 5)

b. Maximum try factor (20 ; 25 ; 29)

c. Maximum valid factor (3 : 4 : 5)

d. Initial Temperature (0.50 ; 0.55 ; 0.60 ; 0.62 ; 0.65 ; 0.70 ; 0.75)

e. Cooling Rate (0.90 ; 0.91 ; 0.92 ; 0.93 ; 0.94 ; 0.95 ; 0.97 ; 0.98)

f. Mixture Rate (Cut-or-Combine/Reshape) (10%/90% ; 5%/95% ;4%/96% ; 3%/97% ; 2%/98%)

The effects of temperature dependent mixture rates are also investigated. Using the current temperature, the mixture rate is changed as the time changes. This approach did not give better results than the constant mixture rates.

There are almost 8,000 combinations of these parameters. However, this is too many to evaluate experimentally. So, the following process is used to find a near-optimum combination. At the beginning of the search, all values of "maximum frozen factor" (3; 4; 5) are tested. During these tests, the remaining parameters (b, c, d, e, f) are chosen to be the same as with Reshape in Table 2.1 (0.7; 4; 5; 0.93; 25), except that a mixture rate of 5%/95% is used. From these tests, the best value (5) is chosen. Keeping this value, the second parameter (maximum try factor) is searched and the best result (25) chosen. The rest of the search is done this way in the order of given above for the parameters. After the first pass down to the last

parameter, another value is chosen for the third parameter and the search proceeds as before. In this way, 70 passes were performed for each function across six parameters.

Some of the results of these tests are given in Table 2.3. These are discussed in Chapter IV.

IV. EXPERIMENTAL RESULTS

The number of product terms and the computation time are the two criteria to judge a minimization algorithm. In Fig. 4.1, twelve algorithms are compared over a set of ten 4-valued 4-variable functions each having 200 minterms. The number of

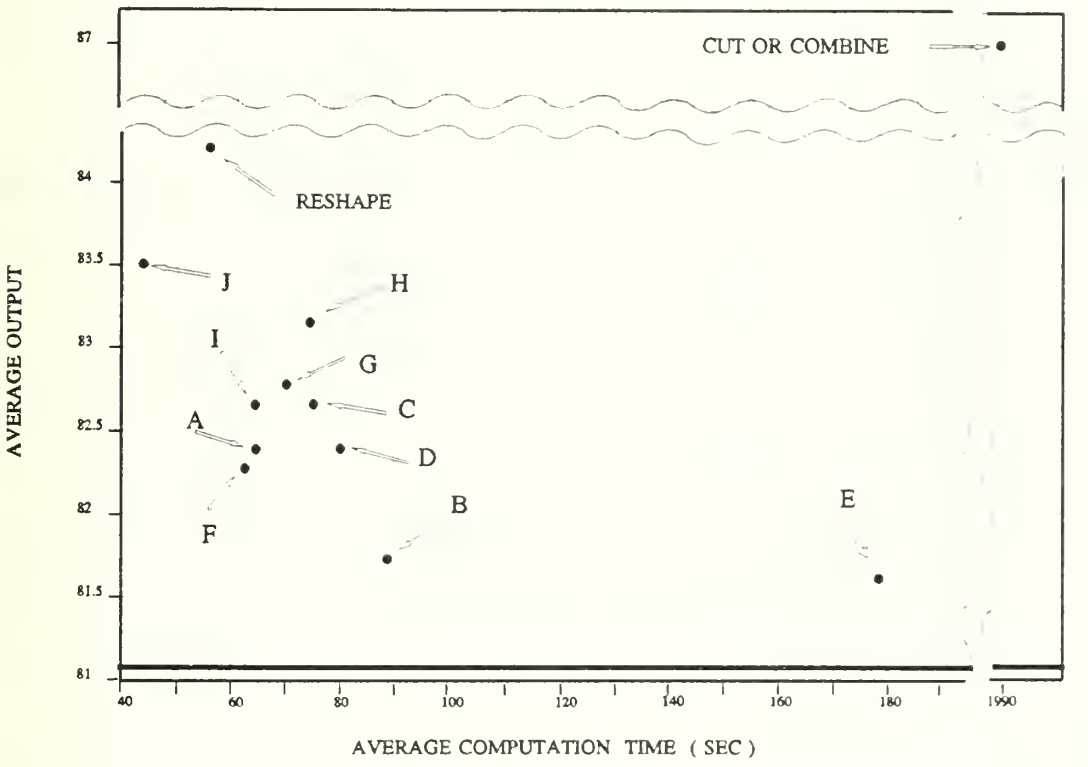


Figure 4.1: Test comparison.

product terms is plotted along the vertical axis, while the average computation time is plotted along the horizontal axis. For example, Reshape in Fig. 4.1 labels a simulated annealing experiment using Reshape with an average number of product

terms of 84.2 and an average computation of 58.0 seconds (the column RSAV in Table 2.1). When Reshape is replaced by the Cut-or-Combine method in a single simulated annealing experiment, we see that the result is a large number of product terms (87.0) with a much longer computation time (1990.6 sec.). This point labelled as Cut-or-Combine in Fig. 4.1. It is far away from all other algorithms.

The point labeled A in Fig. 4.1 represents the experiment discussed in Section II.C that uses the same parameters used in regular Reshape except that the "concurrent multiple path" method is implemented. The result is an average of 82.4 product terms with a computation time of 67.5 seconds.

The point labeled D is an experiment (Table 2.3) that has the same parameters as A. But in this experiment Cut-or-Combine and Reshape moves are mixed in the ratio 4%/96%. The result shows the same average number of product terms (82.4) as in A, but the computation time (81.0 seconds) is worse. This shows that different parameters may improve the result. This is tested as follows.

In the experiment labeled F (Table 2.3), the same mixture rate was used as in D, with new parameters (0.62 initial temperature; 0.91 cooling rate). We see that both the number of product terms and computation time are better than D. This suggests that implementation of mixed annealing requires different parameters than regular Reshape.

In the experiment labeled B (which has the concurrent and mixed method) (Table 2.3), we obtained the best results in terms of product terms with reasonable computation time. This shows that the parameters chosen for experiment labeled B

are close to optimum for concurrent and mixed annealing algorithm. The result is an average of 81.7 product terms with a computation time of 89.9 seconds.

After the good results obtained from experiment labeled B, these same parameters are tested in other cases. At first, these parameters are applied to regular Reshape. In C, which is a regular Reshape with multiple paths (Table 2.3), these same parameters produced worse results than A. This shows that the good results produced by concurrent and mixed annealing are not just due to the choice of parameters.

I is the same as F except that the initial temperature was chosen as 0.7 (as in regular Reshape). The computation time is nearly the same, but average number of product terms is worse.

Applying G and H (Table 2.3) also show how incorrect parameter selection can affect the algorithms performance. In these cases, G is the same as B except that the cooling rate is 0.92 instead of 0.94 and H is the same as G except that the initial temperature is 0.65 instead of 0.60. This suggests that parameter selection is important as a whole.

In applying J (Table 2.3), we intended to get a better average output than regular Reshape, but with a faster computation time. In this test new parameters are used (0.62 initial temperature; 0.92 cooling rate; 20 maximum try factor). Indeed this test gave better results with fewer number of product terms and a shorter computation time. Fig. 4.1 shows that J fell in the left lower side of the Reshape. As expected, the number of product terms (83.5) was not better than 81.7 for B. But

there was a large improvement in computation time to 44.0 seconds compared to 89.9 seconds for B.

Fig. 4.1 also shows that there is a tradeoff between number of product terms and computation time. The experiment labeled B seems the best in terms of average computation time and output. That is why we picked it for "Concurrent Multiple and Mixed Annealing" and used for the comparisons below. The solid line at 81.1 represents the best average output found, from all the tests done (approximately 700) with different parameters. This shows that there is opportunity in parameter and mixture rate selection.

For some of the test results given in Table 2.3 (A, B, I, J), the effects of increasing the number of processors are also investigated. In Fig. 4.2 we can see the average number of product terms and the computation times, as a function of number of processors used for concurrent multiple and mixed annealing. Figure 5.2 suggests that J and B are the best in taking advantage of having more processors.

Fig. 4.3 shows the comparison between the regular Reshape and Concurrent Multiple and Mixed Annealing for all the functions (C1,..,C10). This figure also includes the best results known. For five out of the ten cases, Concurrent Multiple and Mixed Annealing does nearly as well as the best known results. Notice that Reshape is significantly worse for the same functions. In all cases, Concurrent Multiple and Mixed Annealing does better than Reshape.

In Table 4.1 below, the comparison of seven heuristics are given. To be able to get a fair comparison, all the heuristics are compared with the Pomper &

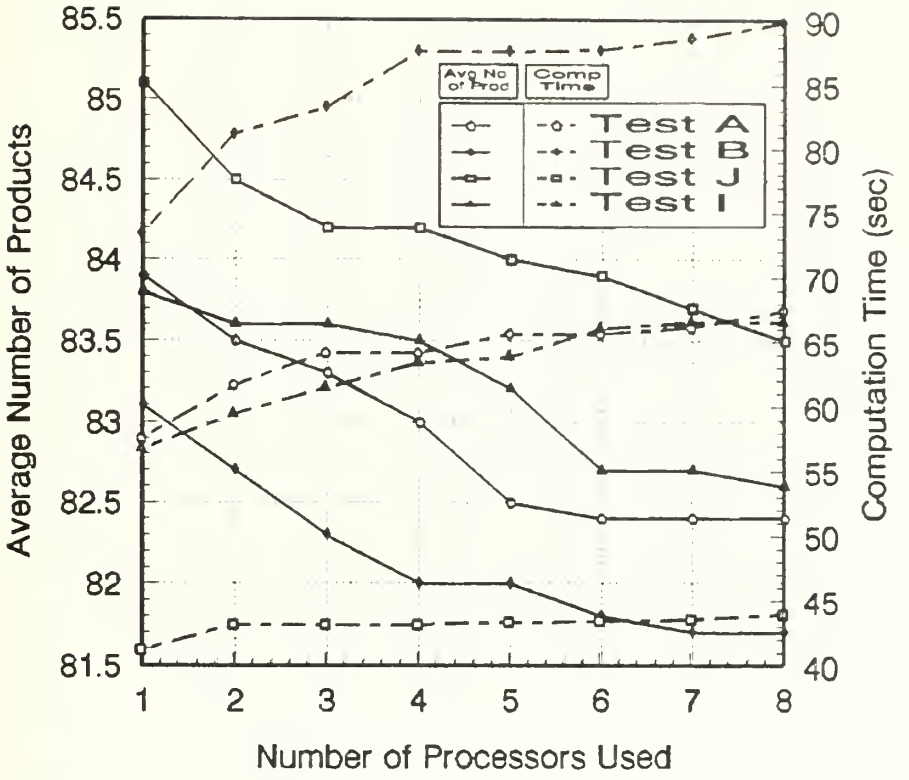


Figure 4.2: Effects of increasing number of processors on average number of products and computation time.

Armstrong [Ref. 5] heuristic. The right column gives the improvement in the average number of product terms divided by the penalty for computation time. As a reference, the results of the experiment labeled J are also included in this table in last row. The outputs of "Concurrent Multiple and Mixed Annealing" are better than the others. The increase in computation time is quite reasonable when we compare the (improvement in average output) / (penalty for computation time) rates for all heuristics.

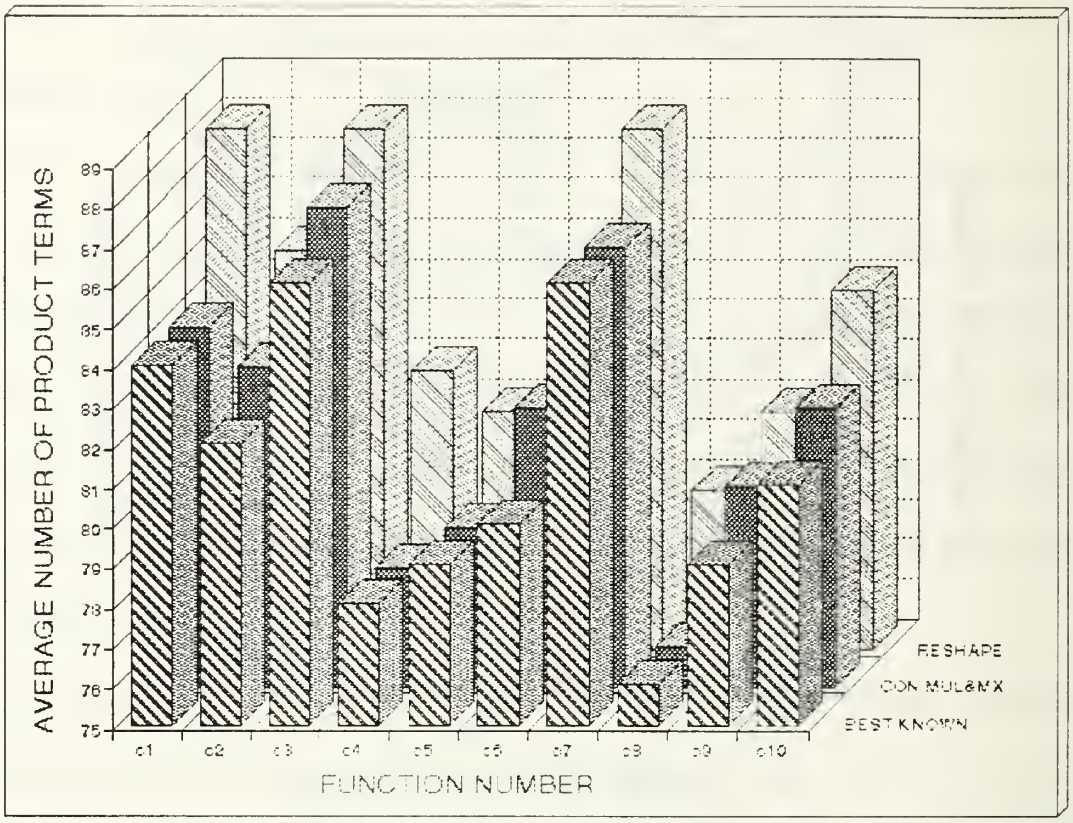


Figure 4.3: "Reshape". "Concurrent multiple and mixed" and "best known" comparison.

TABLE 4.1: HEURISTIC COMPARISON.

HEURISTIC	AVRG PRODUCT TERM	IMPROVEMENT IN AVRG. OUTPUT (%)	AVRG. COMPUT. TIME (sec)	PENALITY IN AVRG. COMPUT. TIME (%)	*IMPRV. IN AVG OUTPUT (%) PENALTY IN COMP. TIME (%)*
POMPER & ARMSTRONG	96.0	0.00 %	8.3	0.00 %	-
DUECK & MILLER	94.0	2.08 %	9.7	16 %	0.1300
YANG & WANG	92.0	4.10 %	30.5	266 %	0.0154
CUT-OR-COMBINE	87.0	9.30 %	1990.6	23900 %	0.0004
RESHAPE	83.9	12.60 %	57.6	576 %	0.0218
CONCURRENT MULTIPLE & MIXED	81.7	14.90 %	89.3	966 %	0.0154
PARAMETERS GROUP J	83.5	13.00 %	44.0	426 %	0.0300

V. CONCLUSIONS

The results and analysis of the tests show significant promise for the Concurrent Multiple and Mixed Simulated Annealing. New approaches to Cut-or-Combine and Reshape heuristics provided better performance than each heuristic alone.

Mixing two moves, Cut-or-Combine and Reshape, in the same simulated annealing experiment gave interesting results. For example, the mixing of a small number of Cut-or-Combine moves (4%) with Reshape moves (96%) allows a minimal solution to be found for a special function, which would be impossible if 100% of the moves were Reshape. The benefit of mixing moves was also shown by our experiments on sets of functions where there were a low average number of product terms (e.g. B and D in Fig. 4.1).

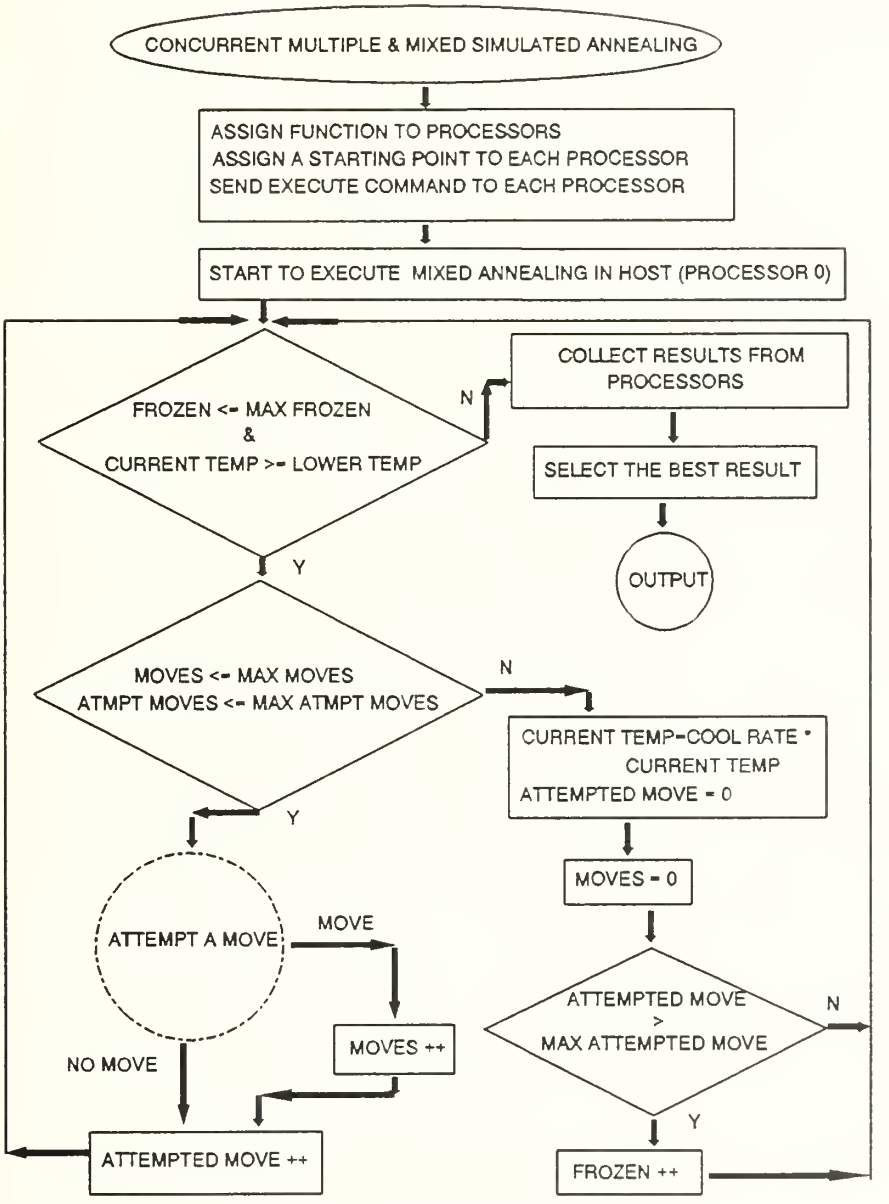
In addition to mixing two different moves, the multiple paths approach for a simulated annealing experiment provided an additional benefit. Performing simulated annealing experiments on a set of ten functions using eight different paths gave better outputs in terms of number of product terms. This experiment yields 82.4 product terms versus 84.2, the expected number for one experiment. However, this reduction had a cost, a large increase in time, 470.5 seconds for the better result versus 58.0 seconds for the worse.

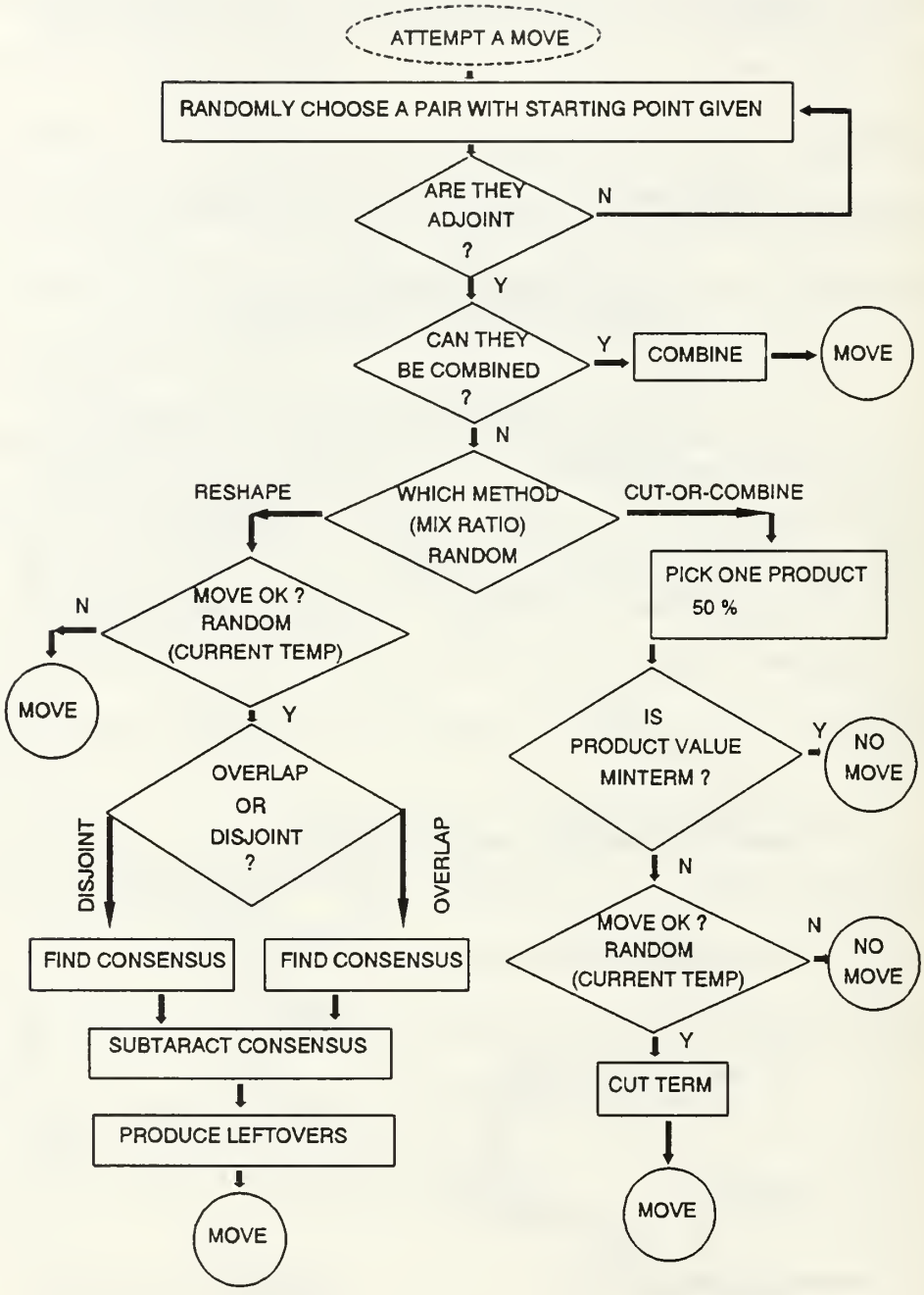
The solution to the large computation time for the multiple path approach was to run multiple experiments concurrently on independent processors. This improved the computation time considerably. With eight processors, there is the prospect of a speedup of 8 over a single processor sequentially performing 8 experiments. The speedups were found on the order of 7, indicating a diversity in computation times over the 8 experiments (speedups of 8 are achievable only if all experiments require identical computation time.)

Finally, the average number of product terms produced by each of 12 experiments are compared with the average number of product terms associated with the best known results. For all experiments, a best result for each function is achieved; the average of these represents the best known realization. The 12 experiments produced, out of approximately 81.1 product terms, a value higher by between 0.5 and 5.9 than this best value.

There is clear advantage in using multiple processors. But, it is also clear that there is a point of diminishing returns in using multiple processors. Our experience suggests that at the eight processors used here, we are beyond that point. However, our results also suggests that this is a fruitful area of research.

VI. APPENDIX A: FLOW DIAGRAM OF THE ALGORITHM




ATTEMPT A MOVE
RANDOMLY CHOOSE A PAIR WITH STARTING POINT GIVEN
ARE THEY ADJOINT ?
N
Y
CAN THEY BE COMBINED ?
Y
COMBINE
MOVE
N
WHICH METHOD (MIX RATIO) RANDOM
RESHAPE
CUT-OR-COMBINE
MOVE OK ? RANDOM (CURRENT TEMP)
N
MOVE
Y
OVERLAP OR DISJOINT ?
DISJOINT
OVERLAP
FIND CONSENSUS
FIND CONSENSUS
SUBTARACT CONSENSUS
PRODUCE LEFTOVERS
MOVE
PICK ONE PRODUCT 50 %
IS PRODUCT VALUE MINTERM ?
Y
NO MOVE
N
MOVE OK ? RANDOM (CURRENT TEMP)
N
NO MOVE
Y
CUT TERM
MOVE

V. APPENDIX B: C CODE UTILIZED

Enclosed in this appendix are the two C programs for Concurrent Multiple and Mixed heuristic in conjunction with HAMLET [Ref. 16]. Each program contains routines that are used by the this heuristic.

1. C code for annealing control:

```
/* $Source: cc.c $
 * $Revision: 2.0 $
 * $Date: 92/10/07 20:12:25 $
 * $Author:  Yildirim $
 * "modifications to original program of yurchak and earle/dueck"
 */

/*************************************************************************
     CC.C   - This module implements the Concurrent Multiple and Mixed using
Cut and Combine heuristic for minimizing an expression.
*************************************************************************/

#include "defs.h"
#include <math.h>
char *isdone();
double a_temp,enumber;
int valid = 0, exhaust_adj = 0, trace =0;
int count_ab,count_cb,count_dv[4],count_ov;
extern int R_flag;
extern int Y_flag;
long a;
extern int node_number;

Cut_Combine()
{
      int    num_impl = 0,j,min_term,absolute_min,max_term,cost;
          int     count,sum,frozen_count = 0,try_count;
      int    *X;
      Implicant        *I;
      extern     double       min_temp,max_temp,cool_rate;
      extern     int    max_frozen,max_try_count,max_valid_count;
```

```
        long    init_cpu.clock();
        double  tot_cpu;
        FILE    *fp;

/* Open output statistic file and initial clock counter for CPU time comparisons. */
        fp = fopen("stats.out","a");
        init_cpu = clock();
        if (E_final[C_C].I != NULL)
            dealloc_expr(&E_final[C_C]);
        HEUR = C_C;
        dup_expr(&E_work.&E_orig);
        /* set parameters     */
        if (R_flag) {
            if (max_valid_count == 0)
                max_valid_count = minterms()*MAX_VALID_FACTOR_R;
            if (max_try_count == 0)
                max_try_count = max_valid_count * MAX_TRY_FACTOR_R;
            if (cool_rate == 0.0)
                cool_rate = COOL_RATE_R;
            }
        else {
            if (max_valid_count == 0)
                max_valid_count = minterms()*MAX_VALID_FACTOR_C;
            if (max_try_count == 0)
                max_try_count = max_valid_count * MAX_TRY_FACTOR_C;
            if (cool_rate == 0.0)
                cool_rate = COOL_RATE_C;
            }

        if((E_work.I=(Implicant*)realloc(E_work.I,sizeof(Implicant)*MAX_TERM))
== NULL)
            fatal("alloc_implicant(): out of memory\n");
        if (!vverify()) printf("we are in big trouble!\n");
        E_final[HEUR].nterm = 0;
        E_final[HEUR].radix = E_orig.radix;
        E_final[HEUR].nvar = E_orig.nvar;
        E_final[HEUR].I = NULL;
        resource_used(START);
count = 0;
tot_cpu = 0.0;
absolute_min = 1000000;
a_temp = max_temp;
```

```
printf("max_temp = %5.2f cool_rate = %5.3f min_temp = %5.3f\nmax_frozen =
%d ",
      max_temp,cool_rate,min_temp,max_frozen);
printf("max_try_count = %d max_valid_count = %d\n",
      max_try_count,max_valid_count);

while ((a_temp > min_temp)&&(frozen_count < max_frozen)) {
      count_cb = 0;
      for(j = 0; j < 4; j++) count_dv[j] = 0;
      count_ov = 0;
      count_ab = 0;
      valid = 0;
      max_term = 0;
      min_term = 1000000;
      try_count = 0;
      enumber = exp(-1.0/a_temp);
      while((try_count < max_try_count)&&(valid < max_valid_count)) {
            pick_a_pair();
            if (a_temp < 0.00){
                  trace = 1;
                  printf("nterm = %d\n",E_work.nterm);
            }
                  if (E_work.nterm< min_term)
                  min_term = E_work.nterm;
            if (E_work.nterm> max_term)
                  max_term = E_work.nterm;

            try_count++;
      }
      if (absolute_min > min_term)
            absolute_min = min_term;
      if ((valid >= max_valid_count) && (min_term < max_term))
            frozen_count = 0;
      else
            frozen_count++;
      tot_cpu = tot_cpu + (clock() - init_cpu)/1000.0;
      init_cpu = clock();
      printf(" %7.3f",a_temp);
      printf(" %3d %3d %3d %6d",min_term,E_work.nterm,max_term,try_count);
      printf(" %10.3f\n",tot_cpu/1000.0);
      a_temp = cool_rate  * a_temp;
      }
      resource_used(STOP);
```

```
        dup_expr(&(E_final[C_C]),&E_work);
        dealloc_expr(&E_work);
        tot_cpu = tot_cpu + (clock() - init_cpu)/1000.0;
        printf("cpu time used = %10.3f sec.\n",tot_cpu/1000.0);
        fprintf(fp,"%5.3f %4d *%4d %10.3f node# %d\n",cool_rate,E_orig.nterm,
                absolute_min,tot_cpu/1000.0,node_number);
        fclose(fp);
}

print_map2()
/*************************************************************************
        :function:
                - Print the Karnaugh map of E_work in its present state
*************************************************************************/
{
        register  i,j;
        int   X[MAX_VAR+2];
        int   *V;

        for (i=0; i < nvar; i++) X[i] = 0;
        for (i=0; i < nvar;) {
                V = eval(&E_work,X);
                printf("%s%3d%c",X[i]==0?"  ":"",V[EVAL],V[HLV]?'.':' ');
                X[i]++;
                for (;i < nvar;) {
                    if (X[i] >= radix) {
                            X[i] = 0;
                            i++;
                            X[i]++;
                    }
                    else {
                            i = 0;
                            break;
                    }
                }
        }
        printf("\n");
}

int vverify()
/*************************************************************************
Verify that the integrity of the function is maintained
*************************************************************************/
```

```
{
	register	i,j;
	int	X[MAX_VAR+2];
	int	*V;
	int	first,second;

	for (i=0; i < nvar; i++) X[i] = 0;
	for (i=0; i < nvar;) {
		V = eval(&E_work,X);
		first = V[EVAL];
		V = eval(&E_orig,X);
		second = V[EVAL];
		if (first != second) return(0);
		X[i]++;
		for (;i < nvar;) {
			if (X[i] >= radix) {
				X[i] = 0;
				i++;
				X[i]++;
			}
			else {
				i = 0;
				break;
			}
		}
	}
	return(1);
}

/*****************************************************************************
	find the sum (truncated) of all minterms in E_work
*****************************************************************************/
int sum_E_work()
{
	register	i,j;
	int	X[MAX_VAR+2];
	int	*V;
	int	result;

	for (i=0; i < nvar; i++) X[i] = 0;
	result = 0;
	for (i=0; i < nvar;) {
		V = eval(&E_work,X);
```

```
            result = result + V[EVAL];
            X[i]++;
            for (;i < nvar;) {
                if (X[i] >= radix) {
                        X[i] = 0;
                        i++;
                        X[i]++;
                }
                else {
                        i = 0;
                        break;
                }
            }
        }
        return(result);
}

/*****************************************************************************
        find the number of minterms in E_work
*****************************************************************************/
int minterms()
{
        register  i,j;
        int   X[MAX_VAR+2];
        int   *V;
        int   result;

        for (i=0; i < nvar; i++) X[i] = 0;
        result = 0;
        for (i=0; i < nvar;) {
            V = eval(&E_work,X);
            result = result + (V[EVAL] >= 1);
            X[i]++;
            for (;i < nvar;) {
                if (X[i] >= radix) {
                        X[i] = 0;
                        i++;
                        X[i]++;
                }
                else {
                        i = 0;
                        break;
                }
```

```
            }
        }
        return(result);
}

/*****************************************************************************
        find the sum (not truncated) of all minterms in E_work
*****************************************************************************/
int     over_sum_E_work()
{
        int i,j,result,temp;

        result = 0;
        for(i = 0; i < E_work.nterm; i++) {
                temp = E_work.I[i].coeff;
                for (j = 0; j < nvar; j++ )
                        temp = temp * (E_work.I[i].B[j].upper -
                                E_work.I[i].B[j].lower +1);
                result = result + temp;
        }
        return(result);
}
```

2. C code for operation control:

```
/* $Source: reshape.c $
 * $Revision: 2.0 $
 * $Date: 92/10/07 20:12:25 $
 * $Author:  Yildirim $
 * "modifications to original program of yurchak and earle/dueck"
 */

/*************************************************************************
      This module controls the operations being conducted on the product terms. It
also controls the concurrency.
*************************************************************************/
Multiple_Mixture()
{
extern long a;
extern int node_number;
extern char if_name[];
static char command[70];
static char processor[8] [6].res_out[26];
static int node_n,node_nn,dig_n,out_n[8],min_out_n;
static float out_t[8],max_out_t;
static int max_node=8;
static char proc_id[6];
static int ten_p[]={1,10,100,1000};
static float tenth_p[]={1.0,0.1,0.01,0.001};

/*  names of the workstations used in the system    */
strcpy(processor[1],"sun22");
strcpy(processor[2],"sun6");
strcpy(processor[3],"sun8");
strcpy(processor[4],"sun9");
strcpy(processor[5],"sun10");
strcpy(processor[6],"sun11");
strcpy(processor[7],"sun17");
/*  different seeds for each random number generator of the processors  */
if(node_number==7) {a=101301;}
if(node_number==6) {a=111001;}
if(node_number==5) {a=109001;}
if(node_number==4) {a=100701;}
if(node_number==3) {a=100501;}
if(node_number==2) {a=100301;}
if(node_number==1) {a=101001;}
```

```
if(node_number==0) {a=100001;

   for(node_n=1;node_n < max_node ;node_n++) {
       strcpy(proc_id.processor[node_n]);
       sprintf(command, "rsh %s 'cd node%d | anneal2 -Y%d %s >> hoho%d ' &
" ,proc_id,node_n, node_n, if_name,node_n);
       system(command);
       }
  }
R_flag++;
Y_flag--;
Cut_Combine();
if(node_number==0) {
       fprintf(stderr,"RESULTS\n");
       min_out_n=9999;
       max_out_t=1.0;
       for(node_n=0;node_n <= (max_node-1);node_n++)
       {
       strncpy(res_out,isdone(node_n),23);
       printf("\n result: %s",res_out);
       for(dig_n=0;dig_n<4;dig_n++) {

out_n[node_n]=out_n[node_n]+(res_out[dig_n]-'0')*ten_p[3-dig_n]*(res_out[dig_
n]!=' ');
             }
       if(out_n[node_n] < min_out_n) min_out_n=out_n[node_n];
       printf("\n min out: %4d",min_out_n);
       for(dig_n=7;dig_n<11;dig_n++){
             out_t[node_n]=out_t[node_n]+(res_out[dig_n]-'0')      *ten_p[10-dig_n]
*(res_out[dig_n]!=' ');
             }

       for(dig_n=12;dig_n<15;dig_n++){
             out_t[node_n]=out_t[node_n]+(res_out[dig_n]-'0')  *tenth_p[dig_n-11];
             }
       if(out_t[node_n] > max_out_t) max_out_t=out_t[node_n];
       printf("\n max time=%4.3f",max_out_t);
       }
       fprintf(stderr,"ALL DONE\n");
}
return;
}
```

```
/*************************************************************************
      Check if the processor finished its assignment
*************************************************************************/
char *isdone(s_no)
int s_no;
{
static int i,done,r_out,chn;
static char ssn[20];
char path_name[19],ch_out[26];
FILE *ifp;
done=0;
sprintf(path_name,"../node%d/stats.out",s_no);
while(done == 0) {
ifp =fopen(path_name, "r");
i=fgetc(ifp);
      while (i != EOF) {
            if (i == 'e') {
                done=1;
                }
            i=fgetc(ifp);
            }
      fclose(ifp);
}
ifp=fopen(path_name, "r");
i=fgetc(ifp);
      while(i != EOF) {
            if(i=='*') {
                for(chn=0;chn<26;chn++) {
                      ch_out[chn]=fgetc(ifp);
                      }
                ch_out[26]=NULL;
                break;
                }
      i=fgetc(ifp);
      }
fclose(ifp);
fprintf(stderr,"node%d done\n",s_no);
strcpy(ssn,ch_out);
return(ssn);
}
```

```
/*************************************************************************
    Select to conduct reshape algorithm
*************************************************************************/
#include "defs.h"
int valid;
extern int count_ab,count_cb,count_ov,count_dv[4],exhaust_adj,trace;
extern double enumber;
extern int R_flag;
extern long a;

pick_a_pair()
{
   int success,temp1,temp2,nterm,j,count,i;
   double d_random();
   success = 0;
   count = 0;
   while ((success == 0) && (count < 100))
   {
     temp1 = random(E_work.nterm);
     temp2 = random(E_work.nterm);
     while (temp1 == temp2)
     {
          temp2 = random(E_work.nterm);
     }

     if ((temp1 < 0) || (temp1 >= E_work.nterm) ||
       (temp2 < 0) || (temp2 >= E_work.nterm)){
          printf("alarm!! %d %d\n",temp1,temp2);
     }
     if (IsAdj(&E_work.I[temp1],&E_work.I[temp2]))
     {
if(trace)
{
   printf("%d adj1=",temp1);
   PrintImp(&E_work.I[temp1]);
   printf("%d addr= %d adj2=",temp2,&E_work.I[temp2]);
   PrintImp(&E_work.I[temp2]);
}
          if (combine(&E_work.I[temp1],&E_work.I[temp2])<0)
            {
              c_subtract(temp2);
            }
            success = 1;
```

```
        }
      else
      {
            count++;
      }
     }
return(0);
}

/*****************************************************************
            subroutine to test for adjacency
*****************************************************************/
IsAdj(imp1,imp2)
      Implicant *imp1,*imp2;
{
      int used,v_index,bprime,aprime;

          used = 0;
          for (v_index = 0; v_index <  nvar;)
              {
               if (((*imp1).B[v_index].lower) >
                  ((*imp2).B[v_index].lower))
                   {
                    aprime = ((*imp1).B[v_index].lower);
                   }
               else
                   {
                    aprime = ((*imp2).B[v_index].lower);
                   }
               if (((*imp1).B[v_index].upper) <=
                  ((*imp2).B[v_index].upper))
                   {
                    bprime = ((*imp1).B[v_index].upper);
                   }
               else
                   {
                    bprime = ((*imp2).B[v_index].upper);
                   }
               if (bprime >= aprime)
                   {
                    v_index++;
                   }
               else if ((aprime == (bprime + 1)) && (used !=1))
```

```
                {
                 v_index++;
                 used = 1;
                }
            else
                {
                 v_index = nvar + 5;
                }
          }
return(v_index ==nvar);
}

/*************************************************************************
        subroutine to select random numbers between 0 and nterm
*************************************************************************/
double d_random ()
  {
    a = (a*125)%2796203;
    return((double)a/2796203);
  }

/*************************************************************************
        subroutine to select random numbers between 0 and nterm
*************************************************************************/
random (nterm)
  int nterm;
  {
  double d_random();
    return((int)(d_random()*nterm));
  }

/*************************************************************
   A subroutine to combine two product terms if possible and to
   randomly chose one of the product terms to be randomly divided.
*************************************************************/
combine(imp1,imp2)
    Implicant *imp1,*imp2;
{
      int p_index,s_index,combcount,bmax,amin,c_var,j,excess,cost;
      double dcost;
      extern double a_temp;
cost = 0;
if (random(25)-1 ==0)
```

```
        R_flag=0;
else
        R_flag=1;
if (IsAbsorb(imp1,imp2))
        {
        cost--;
if (trace)
    printf("inabsorb1\n");
        count_ab++;
        valid++;
        }
else if (IsAbsorb(imp2,imp1))
        {
        cost--;
if (trace)
    printf("inabsorb2\n");
        (*imp1).coeff = (*imp2).coeff;
          for (j=0; j< nvar; j++)
            {
              (*imp1).B[j].lower = (*imp2).B[j].lower;
              (*imp1).B[j].upper = (*imp2).B[j].upper;
            }
        count_ab++;
        valid++;
        }
else if (IsOverlap(imp1,imp2))
   {
        count_ov++;
if (trace)
    printf("inoverlap\n");
      (*imp1).coeff = (*imp1).coeff + (*imp2).coeff;
      if((*imp1).coeff>=radix)
        {
          excess=(*imp1).coeff-(radix-1);
          (*imp1).coeff = radix-1;
        }
      cost--;
      valid++;
   }
else if (IsCombine(imp1,imp2))
   {
        count_cb++;
if (trace)
```

```
    printf("incombine\n");
     for (j=0; j< nvar; j++)
        {
         if (((*imp1).B[j].lower) > ((*imp2).B[j].lower))
             {
               amin   = ((*imp2).B[j].lower);
             }
         else
             {
               amin   = ((*imp1).B[j].lower);
             }
          if (((*imp1).B[j].upper) <= ((*imp2).B[j].upper))
             {
               bmax   = ((*imp2).B[j].upper);
             }
          else
             {
               bmax   = ((*imp1).B[j].upper);
             }
          (*imp1).B[j].lower = amin;
          (*imp1).B[j].upper = bmax;
        }
    cost--;
    valid++;
   }
else if (R_flag) {
       cost = ReshapeCost(imp1,imp2);
       if (trace)
             printf("cost = %d\n",cost);
       if (cost < 1)
             dcost = 0.05;
       else
             dcost = (double) cost;
       enumber = exp(-dcost/a_temp);
       if (d_random() < enumber) {
             count_dv[cost] = count_dv[cost] + 1;
             if (trace)
                 printf("Reshape\n");
             Reshape(imp1,imp2);
             valid++;
             }
       }
else if (d_random() < enumber) {
```

```
        if (random(3) - 1 == 0)
            divide(imp1);
        else
            divide(imp2);
        cost++;
        valid++;
      }
return(cost);
}

/*************************************************************************
                Subroutine to determine if combinable
*************************************************************************/
IsCombine(imp1,imp2)
    Implicant *imp1,*imp2;
{
     int v_index,bprime,aprime,bmax,amin,c_var;
     int used = 0;
        for (v_index = 0; v_index <  nvar;)
            {
            if (((*imp1).B[v_index].lower) >
               ((*imp2).B[v_index].lower))
                {
                aprime = ((*imp1).B[v_index].lower);
                }
            else
                {
                aprime = ((*imp2).B[v_index].lower);
                }
            if (((*imp1).B[v_index].upper) <=
               ((*imp2).B[v_index].upper))
                {
                bprime   = ((*imp1).B[v_index].upper);
                }
            else
                {
                bprime   = ((*imp2).B[v_index].upper);
                }
            if ((((*imp1).B[v_index].lower)==((*imp2).B[v_index].lower))&&
               (((*imp1).B[v_index].upper)==((*imp2).B[v_index].upper))&&
               (((*imp1).coeff)==((*imp2).coeff)))
                {
                v_index++;
```

```
                }
            else if ((aprime == (bprime + 1)) && (used !=1))
                {
                v_index++;
                used = 1;
                }
            else
                {
                v_index = nvar + 5;
                }
            }
return(v_index ==nvar);
}

/*************************************************************************
        Subroutine to determine if complete overlap occurs
*************************************************************************/
IsOverlap(imp1,imp2)
   Implicant *imp1,*imp2;
{
      int v_index;
          for (v_index = 0; v_index <  nvar;)
              {
              if ((((*imp1).B[v_index].lower)==((*imp2).B[v_index].lower))&&
                  (((*imp1).B[v_index].upper)==((*imp2).B[v_index].upper)))
                  {
                  v_index++;
                  }
              else
                  {
                  v_index = nvar + 5;
                  }
              }
return(v_index ==nvar);
}

/*************************************************************************
   Subroutine to do a simple divide on an implicant by variable
*************************************************************************/
cut(imp,c_var,b_cut)
      Implicant *imp;
      int c_var,b_cut;
{
```

```
        int old_up;
        if (trace) {
                printf("in cut Implicant in");
                PrintImp(imp);
        }
        old_up = (*imp).B[c_var].upper;
        (*imp).B[c_var].upper = b_cut;
        add_implicant(imp);
        b_cut++;
        E_work.I[E_work.nterm-1].B[c_var].lower = b_cut;
        E_work.I[E_work.nterm-1].B[c_var].upper = old_up;
        if (trace) {
                printf("in cut Part 1      ");
                PrintImp(imp);
                printf("in cut Part 2      ");
                PrintImp(&E_work.I[E_work.nterm-1]);
        }
}

/*********************************************************************
  Subroutine to do a simple divide on an implicant by coefficient
*********************************************************************/
cutcoeff(imp,c_cut_low,c_cut_high)
     Implicant *imp;
     int c_cut_low,c_cut_high;
   {
     int old_up,p_index,j;
       (*imp).coeff = c_cut_low;
if (trace)
{
    printf("cut_coef_bef");
    PrintImp(imp);
}
       add_implicant(imp);
E_work.I[E_work.nterm-1].coeff = c_cut_high;
if (trace)
{
    printf("cut_coef_af");
    PrintImp(imp);
}
     return(0);
   }
```

```
/*****************************************************************
        Subroutine to randomly divide an implicant
*****************************************************************/
divide(d_imp)
  Implicant *d_imp;
{
 int v_index, i, total_cuts, r_cut, c_count,c_cut_low,c_cut_high;
 int k, kprime, j, list1[100], list2[100];
 total_cuts = 0;
 for(v_index = 0; v_index < nvar; v_index++)
   {
    i = (*d_imp).B[v_index].upper-(*d_imp).B[v_index].lower;
    total_cuts = total_cuts + i;
   }
 if ((*d_imp).coeff == radix - 1)
   {
    i=1;
    for (k=1; k<radix; k++)
      {
       if (k>(*d_imp).coeff - k)
         {
          kprime = k;
         }
       else
         {
          kprime = ((*d_imp).coeff - k);
         }
       for (j= kprime; j<radix; j++)
         {
          list1[i] = k;
          list2[i] = j;
          i++;
         }
      }
    c_count = i-1;
   }
else
   {
     c_count = (*d_imp).coeff/2;
   }
total_cuts = c_count + total_cuts;
if (total_cuts != 0)
  {
```

```
if (trace)
   {
     printf("INDIVIDE");
     PrintImp(d_imp);
   }
r_cut = random (total_cuts) + 1;

if (((*d_imp).coeff == radix-1)&&(r_cut <= c_count))
     {
       c_cut_low = list1[r_cut];
       c_cut_high = list2[r_cut];
       cutcoeff(d_imp,c_cut_low,c_cut_high);
     }
else if (r_cut < (*d_imp).coeff)
     {
       c_cut_low = r_cut;
       c_cut_high = (*d_imp).coeff - r_cut;
       cutcoeff(d_imp,c_cut_low,c_cut_high);
      }
else
      {
       r_cut = r_cut - (c_count);
       i=0;
       while (((*d_imp).B[i].upper - (*d_imp).B[i].lower) < r_cut)
           {
            r_cut = r_cut - ((*d_imp).B[i].upper - (*d_imp).B[i].lower);
            i++;
           }
       r_cut = (*d_imp).B[i].lower + (r_cut-1);
       cut(d_imp,i,r_cut);
      }
 }
  return (0);
}
/*************************************************************************
     Subroutine to determine if one implicant can absorb another
*************************************************************************/
IsAbsorb(imp1,imp2)
      Implicant *imp1,*imp2;
{
      int v_index;
      v_index = 0;
       if((*imp1).coeff == (radix - 1))
```

```
        {
          for (v_index = 0; v_index <  nvar;)
              {
               if ((((*imp1).B[v_index].lower)<=((*imp2).B[v_index].lower))&&
                   (((*imp1).B[v_index].upper)>=((*imp2).B[v_index].upper)))
                    {  v_index++;
                     }
               else  {
                     v_index = nvar + 5; }
              }
         }
return(v_index ==nvar);
}

PrintImp(I)
      Implicant *I;
{     int i;
      printf("+%d",(*I).coeff);
      for (i = 0; i < nvar; i++)
      {     printf("*x%d(%d,%d)",i+1,(*I).B[i].lower,(*I).B[i].upper);
      }
      printf("\n");
}
```

LIST OF REFERENCES

1. K. C. Smith, "The prospect for multivalued logic: a technology and application view, " *IEEE Trans. Computers*, Dec. 1981, pp. 619-632.

2. S. L. Hurst, "Multiple-valued logic - its status and its future," *IEEE Trans. Computers*, Vol C-33, Dec. 1984. pp. 1160-1179.

3. H. G. Kerkhoff "Theory and design of multiple-valued logic CCD's," *in Computer Science and Multiple-Valued Logic* (ed. D. C. Rine),North Holland,New York, 1984. pp. 502-537.

4. J. Butler and H. G. Kerkhoff "Multiple-valued CCD circuits," *IEEE Computer*, March 1988, pp. 58-69.

5. G. Pomper and J. A. Armstrong, "Representation of multivalued functions using the direct cover method," *IEEE Trans. Comp.*, Sep. 1981, pp. 674-679.

6. P. W. Besslich, "Heuristic minimization of MVL functions: a direct cover approach," *IEEE Trans. Comp.*, Vol C-35. Feb. 1986, pp. 134-144.

7. G. W. Dueck and D. Miller, "A direct cover MVL minimization using the truncated sum," *Proc. of 17th Intl. Symp. on MVL*, 1987, pp. 221-226.

8. G. W. Dueck, Algorithms for the minimizations of binary and multiple-valued logic functions, Ph. D. Dissertation, Department of Computer Science, University of Manitoba, Winnipeg, MB. 1988

9. P. Tirumalai and J. T. Butler, "Analysis of minimization algorithms for multiple-valued PLA's," *Proc. of 18th Intl. Symp. on MVL*, 1988, pp. 226-236.

10. E. E. Witte, R. D. Chamberlain, M. A. Franklin,"Parallel Simulated Annealing using speculative computation" *IEEE Trans. Parallel and Distributed Systems*, Vol 2-4,1991, pp. 483-494.

11. G. W. Dueck, R. C. Earle, P. Tirumalai, J. T. Butler, "Multiple-valued programmable logic array minimization by simulated annealing" *Proc. of 22nd Intl. Symp. On MVL*, 1992, pp. 66-74.

12. S. Kirkpatrick, C. D. Gellat, Jr., and M. P. Vecchi, "Optimization by simulated annealing" *Science*, vol. 220, No. 4598, 13 May 1983, pp. 671-680.

13. G. W. Dueck, R. C. Earle, P. Tirumalai, J. T. Butler, "Multiple-valued programmable logic array minimization by simulated annealing" Naval Postgraduate School Technical Report NPS-EC-92-004, Feb 1992.

14. C. Yang and Y. M. Wang, "A neighborhood decoupling algorithm for truncated sum minimization," *Proceedings of the 1990 International Symposium on Multiple-Valued Logic*, May 1990, pp. 153-160.

15. C. Yang and O. Oral, "Experiences of parallel processing with direct cover algorithms for multiple-valued logic minimization," *Proceedings of the 1992 International Symposium on Multiple-Valued Logic*, May 1992, pp. 75-82.

16. J. M. Yurchak and J. T. Butler, "HAMLET - An expression compiler/optimizer for the implementation of heuristics to minimize multiple-valued programmable logic arrays" *Proceedings of the 1990 International Symposium on Multiple-Valued Logic*, May 1990, pp. 144-152.

17. J. M. Yurchak and J. T. Butler, "HAMLET user reference manual," *Naval Postgraduate School Technical Report* NPS-6290-015, July 1990.

INITIAL DISTRIBUTION LIST

		No. of Copies
1.	Defense Technical Information Center Cameron station Alexandria, VA 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
4.	Professor Jon T. Butler, Code EC/Bu Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
5.	Dr. George Abraham, Code 1005 Office of Research and Technology Naval Research Laboratories 4555 Overlook Ave., N.W. Washington, DC 20375	1
6.	Dr. Robert Williams Naval Air Development Center, Code 5005 Warminister, PA 18974-5000	1
7.	Dr. James Gault U.S. Army Research Office P.O. Box 12211 Research Triangle Park, NC 27709	1

8. Dr. Andre van Tilborg 1
Office of Naval Research, Code 1133
800 N. Quincy Str.
Arlington, VA 22217-5000

9. Dr. Clifford Lau 1
Office of Naval Research
1030 E. Green Str.
Pasadena, CA 91106-2485

10. Deniz Kuvvetleri Komutanlığı 1
Personel Eğitim Daire Başkanlığı
Ankara, TURKEY

11. Deniz Harp Okulu Komutanlığı 1
Tuzla İstanbul, TURKEY

12. Gölcük Tersanesi Komutanlığı 1
Gölcük Kocaeli, TURKEY

13. Taşkızak Tersanesi Komutanlığı 1
Kasımpaşa İstanbul, TURKEY

229.546

Thesis
Y473 Yildirim
c.1 Multiple-valued pro-
grammable logic array
minimization by con-
current multiple and
mixed simulated annealing.

Thesis
Y473 Yildirim
c.1 Multiple-valued pro-
grammable logic array
minimization by con-
current multiple and
mixed simulated annealing.

3 2768 00033206 8